Regular Expression Examples

wiki.tcl-lang.org/page/Regular Expression Examples

Regular Expression Examples is a list, roughly sorted by complexity, of regular expression examples. It also serves as both a library of useful expressions to include in your own code.

For advanced examples, see Advanced Regular Expression Examples You can also find some regular expressions on Regular Expressions and Bag of algorithms pages.

See Also

Example Regexes to Match Common Programming Language Constructs

re_syntax

URI detector for arbitrary text as a regular expression

Arts and crafts of Tcl-Tk programming

Regular Expressions

Regular Expression Debugging Tips

Visual Regexp
A **terrific** way to learn about REs.

Redet
Another tool for learning about and working with REs.

Regular Expression Debugging Tips
More tools.

Simple regexp Examples

regexp has syntax:

regexp ?switches? exp string ?matchVar? ?subMatchVar subMatchVar ...?

If *matchVar* is specified, its value will be only the part of the *string* that was matched by the *exp*. As an example:

```
regexp {c.*g} "abcdefghi" matched
puts $matched       ;# ==> cdefg
```

If any *subMatchVar*s are specified, their values will be the part of the *string* that were matched by parenthesized bits in the *exp*, counting open parentheses from left to right. For example:

```
regexp {c((.*)g)(.*)} "abcdefghi" matched sub1 sub2 sub3
puts $matched       ;# ==> cdefghi
puts $sub1          ;# ==> defg
puts $sub2          ;# ==> def
puts $sub3          ;# ==> hi
```

Many times, people only care about the *subMatchVar*s and want to ignore *matchVar*. They use a "dummy" variable as a placeholder in the command for the *matchVar*. You will often see things like

```
regexp $exp $string -> sub1 sub2
```

where ${->} holds the matched part. It is a sneaky but legal Tcl variable name.

PYK 2015-10-29: As a matter of fact, **every** string is a legal Tcl variable name.

Splitting a String Into Words

"How do I split an arbitrary string into words?" is a frequently asked question. If you use split $string { }, then multiple spaces will produce a list with empty elements. If you try to use foreach or lindex or some other list operation, then you must be sure that the string is a well-formed list. (Braces could cause problems.) So use a regular expression like this very simple shorthand for non-space characters:

```
{\S+}
```

You can even split a string of text with arbitrary spaces and special characters into a list of words by using the **-inline** and **-all** switches to regexp:

```
set text "Some arbitrary text which might include \$ or {"
set wordList [regexp -inline -all -- {\S+} $text]
```

Split into Words, Respecting Acronyms

```
set data {Marvels.Agents.of.S.H.I.E.L.D}
regsub -all {(\w{2,})\.} $data {\1 }
```

from Tcl Chatroom, 2013-10-09

Floating Point Number

See Extract Numbers From a String.

Letters

Thanks to Brent Welch for these examples, showing the difference between a traditional character matching and "the Unicode way."

Only letters:

```
^[A-Za-z]+$
```

Only letters, the Unicode way:

```
^[[:alpha:]]+$
```

Special Characters

Thanks again to Brent Welch for these two examples.

The set of Tcl special characters:] [$ { } \:

```
[]][${}\\]
```

The set of regular expression special characters: '] [$ ^ ? + * ( ) | \'

CHARACTER	DESCRIPTION
*	The sub-pattern before '*' can occur zero or more times
+	The sub-pattern before '+' can occur one or more times
?	The sub-pattern before '?' can only occur zero or one time
|	(Alteration) Matches any one sub-pattern separated by '|'s. Similar to logical 'OR'.
()	Groups a pattern
[]	Defines a set of characters, or range of characters [a-z,A-Z,0-9]

```
[]][$^?+*()|\\]
```

I don't understand these examples. Why have [,], and then the rest of the characters inside a [] - that just makes the string have [and] there twice, right?

LV: the first regular expression should be seen like this:

{ ... }
Protect the 9 inner characters.

[...]
Define a set of characters to process.

]
If your set of characters is going to include the right bracket character] as a specific matching character, then it needs to be first in the set/class definition.

[${}
More individual characters.

\

Doubled because when regexp goes to evaluate the characters, it would otherwise treat a single backslash \ as a request to quote the next character, the ending right bracket of the set/class.

The second regular expression is interpreted in a similar fashion. There are more characters because there are more metacharacters.

Also, not all characters are there - where are the period, equals, bang (exclamation sign), dash, colon, alphas that are a part of character entry escapes or classes, 0, hash/pound sign, and angle brackets (< and >)? These special characters all have meta meanings within regular expressions...

LV: Apparently no one has come along and updated the above expression to cover these.

Example posted by KC:

A set containing both angle brackets:

```
[\<\>]
```

newline/carriage return

Could someone replace this line with some verbiage regarding the way one uses regular expressions for specific newline-carriage return handling (as opposed to the use of the $ metacharacter)?

Janos Holanyi: I would really need to build up a re that would match one line and only one line - that is, excluding carriage-return-newline's (\r\n) from matching... How would such a re look like?

LV: how about something like this?

```
% set a "abc
ev"
# a now has two lines in it
% regexp -line -- {(.*)} $a b c d
1
% puts $b
abc
% puts $c
abc
```

If you want to keep carriage returns or newlines by themselves, but not when they are together, you need something like:

```
regexp --  {^([^\r]|\r(?!\n))*}  $a b c d
```

This allows plain carriage return or plain newline.

Thanks to bbh and Donal Fellows for this regular expression.

Back References

From comp.lang.tcl:

I did some experimenting with other strings, like "just a HHHHEEEEAAAADDDDEEEERRRR". The regular expression (.)\1\1\1 does the job I would have wanted, whereas (.){4} will return the last of each four characters - as posted as well.

That surprised me too -- being able to place backreferences within the regex is an extremely powerful technique.

```
regsub -all {(.)\1{3}} $string {\1} result
```

for exactly 4 char repeats, and (.)\1+ for arbitrary repeats.

Whitespace After a Newline

PYK 2019-02-21: How does one capture any whitespace followed by a newline, except for newlines? The key is to use a negative lookahead to match empty space not followed by a newline. That bears repeating: Parenthesis are used to isolate the negative lookahead so that what matches immediately prior is the empty string:

```
set a "one two\n\t \t   \tthree four"
regexp {\n((?:(?!\n)\s)*)} $a -> indent
```

This mechanism is effectively an ***and not*** operator.

bll 2019-02-21: I find:

```
regexp {\n([^\n[:graph:]]*)} $a all indent
```

much easier.

PYK 2019-02-21: That picks up much more than whitespace, so not quite the same thing.

IP Numbers

You can create a regular expression to check an IP address for correct syntax. Note that this regular expression only checks for groups of 1-3 digits separated by periods. If you want to ensure that the digit groups are from 0-255, or that you have a valid IP address, you'll have to do additional (non regexp) work. This code posted to comp.lang.tcl by George Peter Staplin

```
set str 66.70.7.154

regexp "(\[0-9]{1,3})\.(\[0-9]{1,3})\.(\[0-9]{1,3})\.(\[0-9]{1,3})" $str all first
second third fourth

puts "$all \n $first \n $second \n $third \n $fourth \n"
```

The above regular expression matches any string where there are four groups of 1-3 digits separated by periods. Since it's not anchored to the start and end of the string (with ^ and $) it will match any string that contains four groups of 1-3 digits separated by periods, such as: "66.70.7.154.9".

If you don't mind a longer regexp, there is no reason you can't ensure that each group of 1-3 digits is in the range of 0-255. For example (broken up a bit to make it more readable):

```
set octet {(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])}
set RE "^[join [list $octet $octet $octet $octet] {\.}]\$"
regexp $RE $str all first second third fourth ;# Michael A. Cleverly
```

recently on comp.lang.tcl, someone mentioned that http://www.oreilly.com/catalog/regex/chapter/ch04.html#Be_Specific talks about matching IP addresses.

Gururajesh: A Perfect regular expression to validate ip address with a single expression.

```
if {[regexp {(^[2][5][0-5].|^[2][0-4][0-9].|^[1][0-9][0-9].|^[0-9][0-9].|^[0-9].)
([2][0-5][0-5].|[2][0-4][0-9].|[1][0-9][0-9].|[0-9][0-9].|[0-9].)([2][0-5][0-5].|
[2][0-4][0-9].|[1][0-9][0-9].|[0-9][0-9].|[0-9].)([2][0-5][0-5]|[2][0-4][0-9]|[1]
[0-9][0-9]|[0-9][0-9]|[0-9])$} $string match v1 v2 v3 v4]} {puts "$v1$v2$v3$v4"}
else {puts "none"}
```

For 245.254.253.2, output is 245.254.253.2

For 265.254.243.2, output is none, As ip-address can`t have a number greater than 255.

Lars H: Perfect? No, it looks like it would accept 99a99b99c99, since . will match any character. Also, it can be shortened significantly by making use of {4} and the like (see Regular expressions).

Better is

```
if {[regexp {^((([2][5][0-5]|([2][0-4]|[1][0-9]|[0-9])?[0-9])\.){3})([2][5][0-5]|
([2][0-4]|[1][0-9]|[0-9])?[0-9])$} $IP $string match v1 v2 v3 v4]} {puts
"$v1$v2$v3$v4"} else {puts "none"}
```

Tcllib should be useful

- http://docs.activestate.com/activetcl/8.5/tcllib/dns/tcllib_ip.html
- http://tcllib.sourceforge.net/doc/tcllib_ip.html

freethomas: I thinks this regexp is much simple and easier for IP number

```
set str "66.70.7.154"
regexp {(\d+)(\D)(\d+)(\D)(\d+)(\D)(\d+)} $str match
```

AMG: This expression allows any character to separate the octets, not just period. I sincerely doubt this is what you want. Use \. instead of \D. Also it's not anchored with ^ and $, so it works on substrings rather than requiring that the whole string match. Though maybe this is what you want since you explicitly capture the matching substring.

I already fixed the syntax issue of saying { at the beginning but leaving out the closing }, also of leaving out the first (.

I see no reason to use (and) grouping. You don't give variables into which the subexpressions would be captured, and it's pointless to capture the dots between the octets. (See what I did there?) Try this:

```
regexp {^\d+\.\d+\.\d+\.\d+$} $str
```

AMG: Here's a very similar script (to Lars H's contribution) that uses scan instead of regexp. It's much more readable, in my opinion.

```
if {[scan $string %d.%d.%d.%d a b c d] == 4
 && 0 <= $a && $a <= 255 && 0 <= $b && $b <= 255
 && 0 <= $c && $c <= 255 && 0 <= $d && $d <= 255} {
    puts $a.$b.$c.$d
} else {
    puts none
}
```

There are a few differences. One, the trailing dot is omitted from the first three output variables (which I call a, b, c, d instead of v1, v2, v3, v4). Two, leading zeroes are permitted and discarded. Three, -0 is accepted as 0. Four, garbage at the end of $string is silently discarded. Five, each octet can have a leading +, e.g. +255.+255.+255.+255. Six, it's *OVER FIVE TIMES FASTER!* On this machine, my version using scan takes 15 microseconds, whereas your version using regexp takes 78 microseconds. Use time to measure performance. (I replaced puts with return when testing.)

Now, here's a hybrid version that uses regexp.

```
if {[regexp {^(\d+)\.(\d+)\.(\d+)\.(\d+)$} $string _ a b c d]
 && 0 <= $a && $a <= 255 && 0 <= $b && $b <= 255
 && 0 <= $c && $c <= 255 && 0 <= $d && $d <= 255} {
    puts $a.$b.$c.$d
} else {
    puts none
}
```

This version takes 46 microseconds to execute. It doesn't accept leading + or -. It rejects garbage at the end of the string. It treats the octets as octal if they are given leading zeroes, and invalid octal is always accepted. The reason for this last is because if treats strings containing invalid octal as nonnumeric text, so the <= operator is used to sort text rather than compare numbers. Corrected version:

```
if {[regexp {^(\d+)\.(\d+)\.(\d+)\.(\d+)$} $string _ a b c d]
 && [string is integer $a] && 0 <= $a && $a <= 255
 && [string is integer $b] && 0 <= $b && $b <= 255
 && [string is integer $c] && 0 <= $c && $c <= 255
 && [string is integer $d] && 0 <= $d && $d <= 255} {
    puts $a.$b.$c.$d
} else {
    puts none
}
```

This version takes 47 microseconds and it rejects invalid octal. However, it still interprets numbers as octal if leading zeroes are given, so 0377.255.255.255 is accepted (but 0400.255.255.255 is rejected). To fix this, it would be necessary to make a pattern that rejects leading zeroes unless the octet is exactly zero, something like: (0|[^1-9]\d*). But this is getting clumsy and slow; I prefer the scan solution. regexp: not always the right tool!

Gururajesh:

```
set string "0377.255.255.255"
if {[regexp {^(\d+)\.(\d+)\.(\d+)\.(\d+)$} $string _ a b c d]
 && [string is integer $a] && [scan $a %d v1] && 0 <= $v1 && $v1 <= 255
 && [string is integer $b] && [scan $b %d v2] && 0 <= $v2 && $v2 <= 255
 && [string is integer $c] && [scan $c %d v3] && 0 <= $v3 && $v3 <= 255
 && [string is integer $d] && [scan $d %d v4] && 0 <= $v4 && $v4 <= 255} {puts
$v1.$v2.$v3.$v4} else {puts none}
```

This will be ok... for above mentioned issue.

AMG: Why call scan four times? A single invocation can do the job:

```
set string "0377.255.255.255"
if {[regexp {^\d+\.\d+\.\d+\.\d+$} $string]
 && [scan $string %d.%d.%d.%d a b c d] == 4
 && 0 <= $a && $a <= 255 && 0 <= $b && $b <= 255
 && 0 <= $c && $c <= 255 && 0 <= $d && $d <= 255} {
    puts $a.$b.$c.$d
} else {
    puts none
}
```

I don't see any drawbacks to this approach. The regular expression is simple and is used only to reject + and - signs and garbage at the end, scan does the job of splitting and converting to integers, and math expressions check ranges. Three tools, each doing what they're designed for.

CJB: Here is a pure regexp version with comparable performance. It matches any valid ip, rejecting octals. However it does not split the integers and is therefore only useful for validation. The timings on my computer were about 22 microseconds for this version compared to 28 microseconds for the regexp/scan combo (I removed the puts statements for the comparison because they are slow and tend to vary).

Note that the pure scan version is still fastest (about 20 microseconds), splits, and has the same rejections (%d stores integers and ignores extra leading 0 characters).

```
set string 123.255.189.255
regexp {^(?:(?:[2][5][0-5]|[1]?[1-9]{1,2}|0)(?:\.|$)){4}} $string match
```

fh 2012-02-13 11:54:30:

To search IP ADDRESS using Regular Expression

```
set  IP "The Interface IP Address is 198.176.17.16 "
regexp {(25[0-5]|2[0-9][0-9]|[0-9]?[0-9][0-9]?)\.{3}(25[0-5]|2[0-9][0-9]|[0-1]?[0-
9][0-9]?) $IP match
```

Domain names

(First shot)

```
^[a-zA-Z]([a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?\.[a-zA-Z]([a-zA-Z0-9-]{0,61}[a-zA-Z0-
9])?(\.[a-zA-Z]([a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)?$
```

This code does NOT attempt, obviously, to ensure that the last level of the regular expression matches a known domain...

Regular Expression for parsing http string

```
regexp {([[^:]]+)://([[^:/]]+)(:([[0-9]]+))} [ns_conn location] match protocol
server x port
```

the above author should remember this is a Tcl wiki, and not an aolserver one, but thanks for the submission ;)

PYK 2016-02-28: In the previous edit, a - character was added to the regular expression, prohibiting the occurrence of - in *scheme* component of a URL. As far as I can tell, - is allowed in the *scheme* component, so I've reverted that change in the expression above.

E-mail addresses

RS: No warranty, just a first shot:

```
^[A-Za-z0-9._-]+@[[A-Za-z0-9.-]+$
```

Understand that this expression is an attempt to see if a string has a format that is compatible with *normal* RFC SMTP email address formats. It does not attempt to see whether the email address is correct. Also, it does not account for comments embedded within email addresses, which are defined even though seldom used.

bll 2017-6-30 E-mail addresses are quite complicated. You must be careful not to reject valid e-mail addresses. For example, % and + characters are valid. Nobody uses the % sign any more as it is not secure. The + character is very useful, but unfortunately, there are a lot of incorrect e-mail validation routines that reject it.

The following pattern will still reject an e-mail of the form user@[ip-address]. No lengths are checked. It does not check that the top-level domain (e.g. .org, .com, .solutions) is valid.

```
set ::emailpat {
^
(  # local-part
  (?:
    (?:
      (?:[^"().,:;\[\]\s\\@]+)   # one or more non-special characters (not dot)
      |
      (?:
        "  # begin quoted string
        (?:
         [^\\"]  # any character other than backslash or double quote
         |
         (?:\\.) # or a backslash followed by another character
        )+   # repeated one or more times
        "  # end quote
      )
    )
    \.   # followed by a dot
  )*    # local portion with trailing dot repeated zero or more times.
  (?:[^"().,:;\[\]\s\\@]+)|(?:"(?:[^\\"]|(?:\\.))+")  # as above, the final
portion may not contain a trailing dot
)
@
(  # domain-name, underscores are not allowed
  (?:(?:[A-Za-z0-9][A-Za-z0-9-]*)?[A-Za-z0-9]\.)+ # one or more domain specifiers
followed by a dot
  (?:[A-Za-z0-9][A-Za-z0-9-]*)?[A-Za-z0-9]     # top-level domain
  \.?           # may be fully-qualified
)
$
}

proc testit { valid testaddr } {
  set rc NG
  if { [regexp -expanded $::emailpat $testaddr emailaddr local domain] } {
    set rc OK
  }
  if { $rc ne $valid } {
    puts "Fail: ($valid) $testaddr"
  } elseif { $valid eq "OK" } {
    puts "ok: $testaddr $local $domain"
  }
}

# valid e-mails
testit OK {[email protected]}
testit OK {[email protected]}
testit OK {internal-quote."*()"[email protected]}
testit OK {[email protected]}
testit OK {[email protected]}
testit OK {[email protected].}
testit OK {[email protected]}
testit OK {"[email protected]"@example.com}
testit OK {"very.(),:;<>[]\".VERY.\"very@\\ \"very\".unusual"@strange.example.com}
testit OK {[email protected]}
testit OK {#!$%&'*+-/=?^_`{}|[email protected]}
testit OK {"()<>[]:,;@\\\"!#$%&'-/=?^_`{}| ~.a"@example.org}
testit OK {" "@example.org}
testit OK {[email protected]}
testit OK {[email protected].}
testit OK {[email protected]}
# invalid tests
testit NG {[email protected]}
testit NG {[email protected]}
testit NG {Abc.example.com}
testit NG {A@b@[email protected]}
testit NG {a"b(c)d,e:f;g<h>i[j\k][email protected]}
testit NG {just"not"[email protected]}
testit NG {this is"not\[email protected]}
testit NG {this\ still\"not\\[email protected]}
testit NG {[email protected]}
testit NG {[email protected]}
testit NG {[email protected]}
testit NG {[email protected]}
testit NG {john.doe@bad_dom.com}
```

Reference: https://en.wikipedia.org/wiki/Email_address#Examples

XML-like data

To match something similar to XML-tags you can use regular-expressions, too. Let's assume we have this text:

```
% set text {<bo>s</bo><it><bo>M</bo></it>}
```

We can match the body of **bo** with this regexp:

```
% regexp "<(bo)>(.*?)</bo>" $text dummy tag body
```

Now we extend our XML-text with some attributes for the tags, say:

```
set text2 {<bo h="m">s</bo><it><bo>M</bo></it>}
```

If we try to match this with:

```
regexp "<(bo)\\s+(.+?)>(.*?)</bo>" $text2 dummy tag attributes body
```

it *won't work* anymore. This is because \\s+ is greedy (in contrary to the non-greedy (.+?) and (.*?)) and that (the one greedy-operator) makes the whole expression greedy.

See Henry Spencer's reply in tcl 8.2 regexp not doing non-greedy matching correctly , comp.lang.tcl, 1999-09-20.

The *correct* way is:

```
regexp "<(bo)\\s+?(.+?)>(.*?)</bo>" $text2 dummy tag attributes body
```

Now we can write a more general XML-to-whatever-translater like this:

1. Substitute [and] with their corresponding \[and \] to avoid confusion with subst in 3.
2. Substitute the tags and attributes with commands
3. Do a subst on the whole text, thereby calling the inserted commands

```
proc xml2whatever {text userCallback} {
    set text [string map {[ \\[ ] \\]} $text]
    # replace all tags with a call to userCallback
    # this has to be done multiple times, because of nested tags
    # match each tag (everything not space after <)
    # and all the attributes (everything behind the tag until >)
    # then match body and the end-tag (which should be the same as the
    # first matched one (\1))
    while {[regsub -all {<(\S+?)(\s+[^\s>].*?)?\s*?>(.*?)</\1>} $text "\[[list
$userCallback \\1 \\2 \\3]\]" text]} {
                puts doop:
                puts $text
                puts {}
        # do nothing
    }
    return [subst -novariables -nobackslashes $text]
}

# is called from xml2whatever with
# element: the xml-element
# attributes: the attributes of xml-element
# body: body of xml-element
proc myTranslate {element attributes body} {
    # Remove the bracket armour added by xml2whatever
    set element [string map {\\[ [ \\] ]} $element]
    set attributes [string map {\\[ [ \\] ]} $attributes]
    set body [string map {\\[ [ \\] ]} $body]

    # map bo - b; it - i (leave rest alone)
    # do a subst for the body, because of possible nested tags
    switch -- $element {
        bo { return "<b>[subst -novariables -nobackslashes $body]</b>"}
        it { return "<i>[subst -novariables -nobackslashes $body]</i>"}
        default { return "<$element$attributes>[subst -novariables -nobackslashes
$body]</$element>" }
    }
}
```

Call the parser with:

```
xml2whatever $text2 myTranslate
```

You have to be careful, though. Don't do this for large texts or texts with many nested xml-tags because the regular-expression-machine is not the the right tool to parse large,nested files efficiently. (Stefan Vogel)

DKF: I agree with that last point. If you are really dealing with XML, it is better to use a proper tool like TclDOM or tDOM.

PYK 2015-10-30: I patched the regular expression to fix an issue where the attributes group could pick up part of the tag in documents containing tags with similar prefixes. The fix is to use whitespace followed by non-whitespace other than > to detect the beginning of attributes. There are other things

Negated string

Bruce Hartweg wrote in comp.lang.tcl: You can't negate a regular expression, but you CAN negate a regular expression that is only a simple string. Logically, it's the following:

- match any single char except first letter in the string.
- match the first char in string if followed by any letter except the 2nd
- match the first two if followed by any but the third, et cetera

Then the only thing more is to allow a partial match of the string at end of line. So for a regexp that matches

```
 any line that DOES NOT have the word ''foo'':

set exp {^([^f]|f[^o]|fo[^o])*.{0,2}$}
```

The following proc will build the expression for any given string

```
proc reg_negate {str} {
    set partial ""
    set branches [list]
    foreach c [split $str ""] {
        lappend branches [format {%s[^%s]} $partial $c]
        append partial $c
    }
    set exp [format {^(%s)*.{0,%d}$} [join $branches "|"] \
        [expr [string length $str] -1]]
}
```

Donal Fellows followed up with:

That's just set me thinking; you can do this by specifying that the whole string must be either not the character of the *antimatch**, or the first character of the antimatch so long as it is not followed by the rest of the antimatch. This leads to a fairly simply expressed pattern.

```
set exp {^(?:[^f]|f(?!oo))*$}
```

In fact, this allows us to strengthen what you say above to allow the matching of any negated regular expression directly so long as the first component of the antimatch is a literal, and the rest of the antimatch is expressible in an ERE lookahead constraint (which imposes a number of restrictions, but still allows for some fairly sophisticated patterns.)

* Anything's better than overloading 'string' here!

JMN 2005-12-22: Could someone please explain what is meant by a 'negated string' here? Specifically - what do the above achieve that isn't satisfied by the simpler:

```
set exp {^(?!(.*foo.*))}
```

Doesn't the following snippet from the regexp manpage indicate that a regexp can be negated? where does(or did?) the 'simple string' requirement come in? - is this info no longer current?

```
 (?!re)
 negative lookahead (AREs only), matches at any point where no substring matching
re begins
```

Lars H: It indeed seems the entire problem is rather trivial. In Tcl 7 (before AREs) one sometimes had to do funny tricks like the ones Bruce Hartweg performs above, but his use of {0,2} means he must be assuming AREs. Perhaps there was a transitory period where one was available but not the other.

Oleg 2009-12-11: If one needs to match any string but 'foo', then the following will do the work:

```
set exp {^((?!foo).*)|^(foo.+)}
```

And in general case when one needs to match any string that is neither 'foo' nor 'bar', then the following will do the work:

```
set exp {^((?!(foo|bar)).*)|^((foo|bar).+)}
```

CRML 2013-11-06 In general case when one needs to match any string that is neither 'foo' nor 'bar' might be done using:

```
set exp {^(?!((foo|bar)$))}
```

AMG: Oleg's regexps confuse me. Translated literally, I read them as "match any string that does not begin with foo (or bar) unless that string has more characters after the foo (or bar)." Very indirect, I must say. CRML's suggestion I like better, though I would drop the extra parentheses to obtain: ^(?!(foo|bar)$). This says, "match any string that does not begin with either foo or bar when immediately followed by end of string." In other words, "match any string that is not exactly foo or bar."

Turn a string into %hex-escaped (url encoded) characters:

e.g. Csan -> %43%73%61%6E

```
regsub -all -- {(.)} $string {%[format "%02lX" [scan \1 "%c"]]} new_string
subst $new_string
```

This demonstrates the power of using regsub together with subst, which is regarded as one of the most powerful ways to use regular expressions in Tcl.

Turn a string into %hex-escaped (url encoded) characters (part 2)

This one makes the result more readable and still quite safe to use in URLs e.g. https://wiki.tcl-lang.org -> http%3A%2F%2Fwiki%2Etcl%2Etk

```
regsub -all -- {([^A-Za-z0-9_-])} $string {%[format "%02lX" [scan \1 "%c"]]}
new_string
subst $new_string
```

nl

Joe Mistachkin

The inverse of the above (not optimized):

```
regsub -all -- {%([0123456789ABCDEF][0123456789ABCDEF])} $string {[format "%c"
0x\1]} new_string
subst $new_string
```

glennj 2008-12-16: It can be dangerous to blindly apply subst to the results of regsub, particularly if you have not validated the input string. Here's an example that's not too contrived:

```
set string {[some malicious command]}
regsub -all {\w+} $string {[string totitle &]} result
subst $result
```

This results in invalid command name "Some". What if $string was [exec format c:]?

See DKF's "proc regsub-eval" contribution in regsub to properly prepare the input string for substitution. Paraphrased:

```
set string {[some malicious command]}
set escaped [string map {\[ \\[ \] \\] \$ \\$ \\ \\\\} $string]
regsub -all {\w+} $escaped {[string totitle &]} result
subst $result
```

which results in what you'd expect: the string "[Some Malicious Command]"

APN I don't follow why all the extra \ are needed in the string map. The following should work just as well?

```
set escaped [string map {[ \\[ ] \\] $ \\$ \\ \\\\} $string]
```

PYK 2016-05-28: Indeed:

```
expr { [list {*}{ [  \\[  ]  \\]  $  \\$ \\ \\\\}]
    eq [list {*}{\[  \\[ \]  \\] \$  \\$ \\ \\\\}]} ;# -> 1
```

Maintain proper spacing when formatting for HTML

DG got this from Kevin Kenny on c.l.t.

```
regsub -all { (?= )} $line {\&nbsp;} line

set line {this is an    example}
regsub -all { (?= )} $line {\&nbsp;} line
set line
```

And the output is:

```
this is an&nbsp;&nbsp;&nbsp; example
```

Tabs require replacement, too:

```
set tabFill "[string repeat \\&nbsp\; 7] "
regsub -all {\t} $line $tabFill line
```

---

glennj: Taken from comp.lang.perl.misc, transform variable names into StudlyCapsNames:

```
set old_vars {VARIABLE_ONE VARIABLE_NUMBER_TWO a_really_long_VARIABLE_name}
set NewVars {}
foreach v $old_vars {
   regsub -all {_?(.)([^_]*)} $v {[string toupper "\1"][string tolower "\2"]} new
   lappend NewVars [subst $new]
}
```

When using ASED's syntax checker you get an error of you don't use the -- option to regexp. Instead of regexp {([^A-Za-z0-9_-])} $string you have to write regexp -- {([^A-Za-z0-9_-])} $string

---

LV: A user recently asked:

I have a string that I'm trying to parse. Why doesn't this seem to work?

```
% set str {Acc No: 12345}
% set num [regexp {.*?(\d+).*} $str junk result]
% puts $result
1
```

It looks to me like the *? causes the subsequent \d+ to also be non-greedy and only match the first hit. Did I figure that out correctly? I presume that we currently don't have a way to *turn off* the greediness item?

Of course, in this simplified problem, one could just drop the greediness and code

```
% set num [regexp {(\d+)} $str junk result]
% puts $result
12345
```

I'll let the user decide if that suffices.

PYK 2019-08-15: See "greediness" at Regular Expressions. In short the greediness of every quantifier is the greediness of its branch, regardless of the default preference of the quantifier. A branch, in turn, picks up its greediness from the first quantifier.

How do you select from two words?

```
% set word "foo"
% set result [regexp {(foo|bar)} match zzz]
% set zzz
can't read "zzz": no such variable
???
```

LES: You got the regexp syntax wrong and tried to match the regular expression with the string "match". There is no "zzz" variable (the actual match variable in your code) because your regular expression does not match the string "match". Try this:

```
% set word "foo"
% set result [regexp {(foo|bar)} $word match zzz]
% set match
```

Note that I could have dropped the "zzz" variable, but left it there as a second match variable, as an exercise to you. You should understand why and what it does if you read the regexp page and assimilate the syntax.

Infinite spaces at start and end

RUJ: Could you match the following pattern of following string: infinite spaces at start and end.

```
% set str "  sjkhf sdhj   "
```

LV: try

```
set rest [regexp {^ +.* +$} $str match]
puts $rest
```

which should have a value of 1 (in other words, it matched). Of course, if those leading and trailing spaces are optional, then change the + to a *.

CRML non greedy or greedy does not give the same result. In the previous example, the .* matches all the string up to the last but one char.

```
set rest [regexp {^ +(.*?) *$} $str match noinfinite]
puts $rest
puts "|$noinfinte|"
set rest [regexp {^ +(.*) *$} $str match noinfinite]
puts $rest
puts "|$noinfinte|"
```

URL Parser

See URL Parser.

Match a "quoted string"

AMG: Adapted from Wibble:

```
proc quoted-string {str} {
    regexp {^"(?:[^\\"]|\\.)*"$} $str
}
```

This recognizes strings starting and ending with double quote characters. Any character can be embedded in the string, even double quotes, when preceded by an odd number of backslashes.

Word Splitting, Respecting Quoted Strings

given some text, e.g.

```
here    is some "quoted     text with   lots    of space"  and    more
```

how to parse it into

```
here is some {quoted     text with   lots    of space} and more
```

```
regexp -all -inline {(?:[^ "]|\"[^"]*\")+}
```

see KBK, #tcl irc channel, 2012-12-02

split a string into n-length substrings

```
regexp -all -inline ".{$n}" $string
```

evilotto, #tcl, 2013-02-07

:) Contest: fast way to chop string in short fixed pieces, comp.lang.tcl, 2004-07-19

At Least 1 Alpha Character Interspersed with 0 or More Digits

```
regexp {[[:alnum:]]*[[:alpha:]][[:alnum:]]*} $string
```

Matching a group of strings

```
regexp -nocase {string1|string2|string3 ....} $string
```

We can match a group of strings or subjects in a single regular expression

Sqlite Numeric Literal

```
regexp {^([[:digit:]]*)(?:\.([[:digit:]]+))?(?:[eE][+-]?([[:digit:]]+))?$|^0x[a-
fA-F]+$} number int mant exp
```

ak - 2017-08-08 03:32:33

Regarding negation of regular expressions.

While the regular expression syntax does not allow for simple negation the underlying formalism of (non)deterministic finite automata does. Simply swap final and non-final states to negate, i.e. complement it.

See for example the grammar::fa package in Tcllib, which provides a complement method. It is implemented in the operations package. As are methods to convert from and to regular expressions.